Inverting Permutations In Place

by

Matthew Robertson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2015

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

In this thesis, we address the problem of quickly inverting the standard representation of a permutation on n elements in place. First, we present a naive algorithm to do it using $\mathcal{O}(\log n)$ extra bits in $\mathcal{O}(n^2)$ time in the worst case. We then improve that algorithm, using a small bit vector, to use $\mathcal{O}(\sqrt{n})$ extra bits in $\mathcal{O}(n\sqrt{n})$ time. Using a different approach, we present an algorithm to do it using $\mathcal{O}(\sqrt{n}\log n)$ extra bits in $\mathcal{O}(n \log n)$ time. Finally, for our main result, we present a technique that leads to an algorithm to invert the standard representation of a permutation using only $\mathcal{O}\big(\log^2 n\big)$ extra bits of space in $\mathcal{O}(n \log n)$ time in the worst case.

Acknowledgements

I would like to thank my supervisor Professor J. Ian Munro for his patience, guidance and everything else. I would also like to thank my colleague and friend Hicham El-Zein for our many invaluable discussions on the topic. And finally, I would like to thank my readers Professor Jonathan Buss and Professor Gordon B. Agnew.

Table of Contents

List of Figures

List of Symbols

$[n]$	The set of integers 0 through $n-1$, inclusive.
$\mathcal{F}$	The Fich et al. leader scan.
$\mathcal{H}$	The "hare" scan.
$\mathcal{P}(n)$	Number of bits required to represent a permutation of length n.
$\mathcal{T}$	The "tortoise" scan.
α	A single cycle in a permutation.
ℓ	The length of a cycle.
$\emptyset$	A sentinel value.
$\lg n$	The logarithm of n to the base 2.
π	An arbitrary permutation.
d	The maximum level, or depth, of a cycle.
e	The base of the natural logarithm.
n	The length of a permutation or array.
x_r	The level leader at level r.
`pi`	The standard representation of a permutation.

Chapter 1

Introduction

A *permutation* π is a one-to-one correspondence between a set and itself. Our interest is in finite sets of size n, so without loss of generality, we can assume the underlying set is $[n] = \{0, 1, \ldots, n-1\}$. The *inverse* of the permutation π, π^{-1}, is the permutation such that, for all i, $\pi^{-1} \circ \pi(i) = i$. An array, $A[0, n-1]$, is *permuted* according to π (or the permutation is *applied*) if element $A[i]$ is moved to position $\pi(i)$ for each i. This has the effect of performing the n simultaneous assignments

$$A[\pi(i)] \leftarrow A[i] \text{ for } i \in [n] \ ,$$

which causes its own difficulties if we are to avoid the use of much extra space.

There are several reasonable ways of representing a permutation, but the most natural or *standard* is as an array `pi` in which `pi`$[i] = \pi(i)$, where each element of `pi` is of size $\lceil \lg n \rceil$ bits. Our focus is on replacing the representation of π in `pi` by the representation of π^{-1}, quickly and in place.

By *in place*, we mean that the algorithm executes by rearranging elements of the input without the use of significant extra space for the elements. In terms of inverting permutations, the input permutation is modified to become its own inverse. We consider an algorithm to be in place if it uses only $o(n)$ extra bits of space. In particular, this definition precludes using even 1 extra bit of auxiliary space per element. Ideally, we want our algorithm to use only a polylogarithmic number of extra bits.

An example of an application for inverting a permutation in place arises in data warehousing [10, 3]. Under specific indexing schemes, the permutation corresponding to the rows of a relation sorted by any given key is explicitly stored. To perform certain joins,

the inverse of a segment of the permutation is precisely what is needed. This permutation occupies a substantial portion of the space used by the indexing structure. Doubling this space requirement, to explicitly store the inverse of the permutation, for the sole purpose of improving the time to compute certain joins is not practical.

1.1 Main Contribution

Our main contribution is a technique that leads to an algorithm to invert the standard representation of a permutation using only $\mathcal{O}\left(\log^2 n\right)$ extra bits of space in $\mathcal{O}(n \log n)$ time in the worst case. Previously known algorithms used either quadratic time or a linear number of extra bits of space [5].

1.2 Thesis Outline

In Chapter 2, we present some background on permutations and other related work. We introduce the standard representation of a permutation and justify its use. We also present previous research on the problem of applying a permutation in place, the inspiration for our research. Most notable is the concept of cycle leader algorithms and in particular, what we call the Fich et al. [5] leader algorithm, which is an algorithm to apply a permutation using only $\mathcal{O}\left(\log^2 n\right)$ extra bits of space in $\mathcal{O}(n \log n)$ time. This is a key building block for our research on inverting permutations in place.

We then present a naive approach to invert a permutation in place in Chapter 3 using techniques from [5]. Unfortunately, we cannot just use the Fich et al. leader method alone to invert a permutation. We show the trade-off between the space and time complexity when using a simple leader method. A natural balance leads to an algorithm to invert the standard representation of a permutation using $\mathcal{O}(\sqrt{n})$ extra bits of space in $\mathcal{O}(n\sqrt{n})$ time.

Chapter 4 presents our new technique of "breaking the cycle." This is the strategy which allows us to use the Fich et al. leader method to invert a permutation. First, we show how to simulate a sentinel value and use it to break the cycle. But this leads to a problem in some cases when restoring the original cycle. We then solve the problem by exploiting the limited number of unique cycle lengths in a permutation. This leads to an algorithm to invert the standard representation of a permutation using $\mathcal{O}(\sqrt{n} \log n)$ extra bits of space in $\mathcal{O}(n \log n)$ time. That is as fast as permuting in place in the worst case,

but uses significantly more space. Finally, for our main result, we show how to break the cycle by creating a small loop in it. This leads to an algorithm to invert the standard representation of a permutation using only $\mathcal{O}\left(\log^2 n\right)$ extra bits of space in $\mathcal{O}(n \log n)$ time in the worst case. This algorithm has the same space and time complexity as the algorithm to apply a permutation in place, `permute`, from [5].

In Chapter 5 we present the conclusion and some ideas for future work. We explain that a better leader method will lead to a better algorithm to invert a permutation in place. Also, we explore the idea of applying arbitrary powers to a permutation in place.

1.3 Computational Model

We use the word RAM model [6, 1] in this thesis. The standard arithmetic operations are supported on words in constant time. The elements of an array can be accessed or mutated in constant time. A word, or pointer, is at least $\lceil \lg n \rceil$ bits. Memory use can be measured in words, pointers, or bits depending on convenience.

Chapter 2

Background and Related Work

There are exactly $n!$ different permutations of length n. Thus the exact number of bits required to represent a permutation is $\mathcal{P}(n) = \lceil \lg n! \rceil$. This is the information theoretic lower bound [10]. The asymptotic space complexity is given by

$$\lceil \lg n! \rceil = n \lg n - n \lg e + \mathcal{O}(\log n) \in \mathcal{O}(n \log n) \quad . \tag{2.1}$$

A permutation of length n can be represented in memory using $n \lceil \lg n \rceil$ bits as an array, `pi`, of the positions 0 through $n-1$. Although there are other succinct representations of permutations, this simple representation matches the space described in (2.1) to about $n \lg e$ bits. The elements of the array can be ordered such that cycles are stored consecutively, or such that the element `pi[i]` stores $\pi(i)$. The latter is a very natural representation, and is the standard representation we are most interested in.

Figure 2.1 shows an example of a permutation in its standard representation and as a graph. The arrows of the graph follow the direction $i \to \pi(i)$. We want to invert the permutation, i.e., reverse all the arrows in the graph.

i	0	1	2	3	4	5	6	7	8	9
$\pi(i)$	4	2	1	0	5	3	8	6	7	9

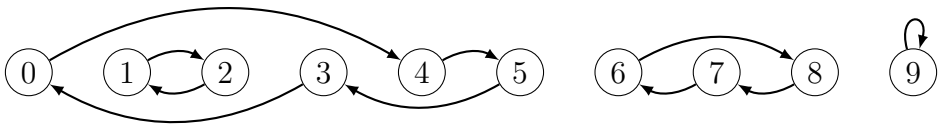


Figure 2.1: An example of a permutation.

A succinct representation is a very space efficient representation that approaches the information theoretic lower bound [9]. Munro et al. [10] present a succinct representation of a permutation using only $\mathcal{P}(n) + o(n)$ bits that can compute $\pi^k(i)$ for any arbitrary power k in $\mathcal{O}(\log n / \log\log n)$ time. They also present a representation taking $(1+\epsilon)n \lg n$ bits that can compute $\pi^k(i)$ in constant time. These representations are static, i.e., do not support mutation rapidly, and therefore do not seem useful for our purposes. We do not know of any representation using less than $n \lg n$ bits that supports mutation rapidly.

The standard representation of a permutation uses between about $n \lg e$ bits and about $n(1 + \lg e)$ bits more than $\mathcal{P}(n)$, depending on the relation between n and a power of 2. This is the consequence of there being no repeated values. However, we can still reduce the space required by this representation by encoding k consecutive elements into a single object. This object is essentially the k digit, base n number $\texttt{pi}[i]\texttt{pi}[i+1]\ldots\texttt{pi}[i+k-1]$. So instead of encoding n objects of size $\lceil \lg n \rceil$ bits, totalling up to $n \lg n + n$ bits; we can encode n/k objects of size $\lceil k \lg n \rceil$ bits, totalling up to $n \lg n + n/k$ bits. To decode a value, we need a constant number of $\lceil k \lg n \rceil$ bit precision arithmetic operations.

A permutation is a collection of disjoint cycles. The cycles in our example permutation can be traversed by following the arrows around and back to the starting position. That is the same as repeatedly evaluating $i \leftarrow \pi(i)$, starting from some position i within the cycle. For example, (0 4 5 3) is a cycle, so is (1 2), and (9) is a singleton, i.e., a cycle by itself. The cycle form of a permutation is each cycle written out explicitly, separated by a delimiter. One way to avoid using a delimiter is to use an n-bit vector to mark the start of each cycle. Since disjoint cycles commute, the cycles can be written in any order. If the cycles are written starting from their minimum value and ordered in decreasing order by this cycle minimum value, there is no need for a delimiter or bit vector. This is a unique representation using $n \lceil \lg n \rceil$ bits.

The cycle form of a permutation is trivial to invert by reversing each cycle. However, this form does not support the operation of computing $\pi(i)$ in constant time. The cycle structure of a permutation can still be exploited using the standard representation.

2.1 Cycle Leader Algorithms

In permuting A according to π, it is not sufficient to simply assign $A[\pi(i)] \leftarrow A(i)$ for each $i \in [n]$, because an element in A may have been mutated before it has been accessed. There are several very simple ways to deal with this: The use of an auxiliary copy of A, use of an n-bit vector to mark moved elements, or by destroying the representation of π by

assigning $\texttt{pi}[i] \leftarrow i$ as it is traversed. None of these are sufficient for our purposes because they are either not in place or they destroy information.

Fich et al. [5] were interested in applying a permutation to an array of objects A when the permutation is given as an oracle $\pi(i)$. To do this they developed the idea of a *cycle leader*, a protocol by which precisely one position in each cycle is so designated. The appropriate process is performed on the data for each cycle by testing each position i for cycle leadership. So in applying a permutation, we rotate each cycle once starting from its cycle leader.

The standard representation of a permutation perfectly represents the oracle described in [5]. A basic operation of permuting is to rotate an array according to a cycle of the permutation. Figure 2.2 shows a method to do this by traversing the cycle, starting from some position within the cycle. That starting position is the leader of the cycle.

```
method rotate(A, leader)
    i ← π(leader)
    while i ≠ leader do
        swap(A[i], A[leader])
        i ← π(i)
```

Figure 2.2: Rotate A according to the cycle containing *leader*.

Identifying a cycle leader is not trivial because the standard representation of a permutation does not naturally distinguish cycles. The permutation can be processed by iterating over the permutation, processing each cycle only on its leader. A cycle can be traversed starting from any position within the cycle, so the only requirements of the leader are that it must be in the cycle, and must be unique. The left most position in the cycle, or *min leader* as shown in Figure 2.3, is a simple example of a leader method. In our example permutation, positions 0, 1, 6, and 9 are cycle leaders according to min leader.

```
method isLeader(leader)
    i ← π(leader)
    while i > leader do
        i ← π(i)
    return i ≟ leader
```

Figure 2.3: Determine if *leader* is the minimum position in the cycle.

Figure 2.4 shows a cycle leader algorithm to permute the array A according to the permutation. The `isLeader` method call can be replaced by any leader method, such as the min leader method described above.

```
method permute(A)
    for i ← 0 to n − 1 do
        if isLeader(i) then
            rotate(A, i)
```

Figure 2.4: Permute A according to the permutation.

The problem is to identify a position as leader by starting at that position and traversing only forward along the cycle. Choosing the min leader would take as few as $2n$ value inspections as we test each position of a permutation consisting of one large cycle in decreasing order; or as many as n^2 value inspections for a large cycle in increasing order. We note that for a random cycle of length n this total cost would be about $n \lg n$, curiously close to the worst case time bound achieved in [5]. The analysis is similar to the bidirectional distributed algorithm for finding the smallest of a set of n uniquely numbered processors arranged in a circle [7]. However, we are only interested in finding algorithms with good performance in the worst case.

The approach of [5] is to designate as leader the position from which such an anomaly as "local minima" can be observed[1]. By local minimum, we mean a position i such that $\pi^{-1}(i) > i < \pi(i)$, but inductively. Given the unidirectional nature of our standard representation, it is easy to compute $\pi(i)$ but not $\pi^{-1}(i)$ from a position i. So instead, we will find a position i such that $\pi(i)$ is a local minimum, i.e., $i > \pi(i) < \pi \circ \pi(i)$. We call this the *Fich et al. leader* method and it is shown in Figure 2.5.

Let $\alpha_0 = \alpha$ be a cycle in π, and $x_0 = i$ be the potential cycle leader, i.e., the not yet excluded position being tested for leadership. Let α_r be the cycle of the the local minima of α_{r-1}, and $x_r = \alpha_{r-1}(x_{r-1})$ be the potential leader at depth r, for $r > 0$. Recurse until an α_d is found to be a singleton, that is until $\alpha_d(x_d) = x_d$. The maximum depth is $d \leq \lceil \lg \ell \rceil$, where ℓ is the length of α. At each level, no more than half the positions can be local minima. The potential leader can be excluded as soon as a depth r is found such that the condition $x_r > \alpha_r(x_r) \leq \alpha_r^2(x_r)$ fails where $0 < r < d$. This is enough to uniquely define a position to be the leader of the cycle and can be used to permute data in $\mathcal{O}(n \log n)$ worst case time using $\mathcal{O}\left(\log^2 n\right)$ extra bits of space [5]. This is a great improvement in speed, min leader albeit does use less space.

[1] Not quite what is done.

```
method isLeader(leader)
    elbow[1] ← leader
    for i ← 1 to ⌈lg n⌉ do
        next(r)
        if elbow[r] > elbow[r − 1] then
            elbow[r] ← elbow[r − 1]
            next(r)
            if elbow[r] > elbow[r − 1] then
                return false
            elbow[r + 1] ← elbow[r]
        else
            return elbow[r] ≟ elbow[r − 1]
```

Figure 2.5: Determine if *leader* is the Fich et al. leader of the cycle.

Figure 2.6 shows the Fich et al. process for the cycle α in $\pi = [6, 8, 9, 4, 2, 7, 1, 0, 3, 5]$ containing position 0. The local minima of α at level r are represented by α_r. The level leaders x_r are marked with overhead arrows.

$$\alpha_0 = (0\ 6\ 1\ \vec{8}\ 3\ 4\ 2\ 9\ 5\ 7)$$
$$\alpha_1 = (0\ 1\ \vec{3}\ 2\ 5)$$
$$\alpha_2 = (0\ \vec{2})$$
$$\alpha_3 = (\vec{0})$$

Figure 2.6: Example of the Fich et al. leader process.

The leader of the cycle in our example is position 8 because $\alpha_0(8) = 3$, $\alpha_1(3) = 2$, $\alpha_2(2) = 0$, and $\alpha_3(0) = 0$ points to itself. Figure 2.7 shows this as a graph. In general, i is the leader of the cycle if $\alpha_d \circ \alpha_{d-1} \circ \ldots \circ \alpha_0(i) = x_d$, where x_d is the minimum position in the cycle. Dolev, Klawe, and Rodeh [4] and Peterson [11] independently discovered the usefulness of this cycle leader in the context of determining the minimum number in a circular arrangement of n uniquely numbered processes in a distributive manner.

The method in Figure 2.8 has the effect of assigning $elbow[r-1] \leftarrow \alpha_{r-1}(elbow[r])$ where *elbow* is an array of size $d+1$. The elements of the array *elbow* store pointers to the positions at the corresponding levels. It is sufficient to store one pointer for each level.

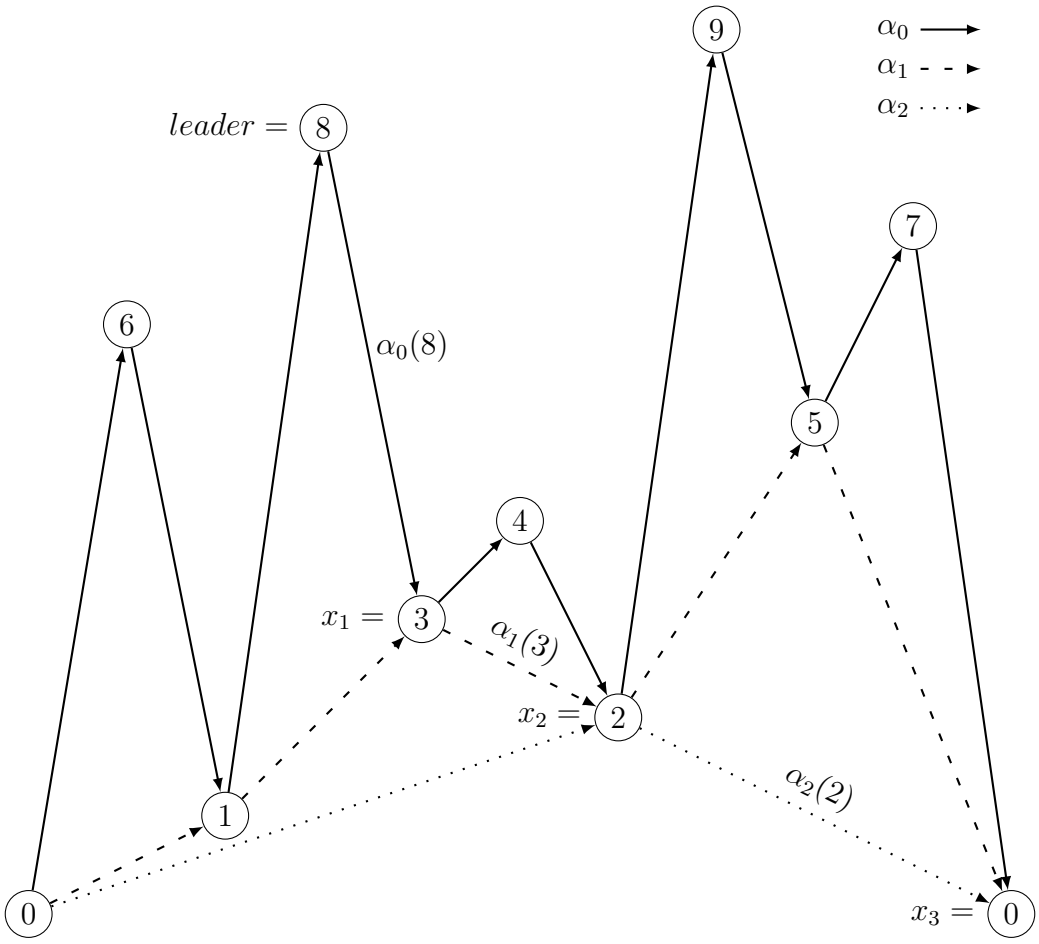


Figure 2.7: Graph of the example Fich et al. leader process.

```
method next()
    elbow[0] ← π(elbow[1])

method next(r)
    if r = 1 then
        next()
    else
        while elbow[r − 1] < elbow[r − 2] do
            elbow[r − 1] ← π(elbow[r − 2])
            next(r − 1)
        while elbow[r − 1] > elbow[r − 2] do
            elbow[r − 1] ← π(elbow[r − 2])
            next(r − 1)
```

Figure 2.8: Determine the next position in α_r.

Theorem 2.1 (Theorem 2.3 of [5])**.** *In the worst case, permuting an array of length n, given the permutation, can be done in $\mathcal{O}(n \log n)$ time and $\mathcal{O}\left(\log^2 n\right)$ additional bits of storage.*

We are interested in inverting the standard representation of a permutation, `pi`, in place; not applying a permutation to an external array, A. The major difference between applying and inverting a permutation is that permuting will process cycles by rotating them on A; whereas inverting will process cycles by reversing them in place, mutating `pi`.

Chapter 3

A Naive Solution

A naive approach to inverting a permutation in place follows the same structure as the `permute` algorithm described in Chapter 2, but reverse the cycles instead of rotating them. This inversion algorithm, `invert`, shown in Figure 3.1, is quite optimistic because reversing a cycle and mutating `pi`, will, in general, change the leader of the cycle. A cycle leader method must be used that will determine the same position to be leader, regardless of whether the cycle has been reversed or not. An example of such a leader method is min leader. Unfortunately, the Fich et al. leader method is not such an example.

```
method invert()
    for i ← 0 to n − 1 do
        if isLeader(i) then
            reverse(i)
```

Figure 3.1: Optimistic algorithm to invert a permutation using cycle leaders.

Reversing a cycle has the same time complexity as rotating a cycle. Figure 3.2 shows how to reverse a cycle by treating the cycle like a linked list.

For the same reasons as presented in Chapter 2 and [5], inverting a permutation using min leader will run in $\mathcal{O}(n^2)$ worst case time and use $\mathcal{O}(\log n)$ extra bits of space. Although this is in place, it is much too slow for our purposes. In this chapter, we present a strategy to improve the speed of using min leader at the cost of more space.

```
method reverse(leader)
    curr ← pi[leader]
    prev ← leader
    while curr ≠ leader do
        next ← pi[curr]
        pi[curr] ← prev
        prev ← curr
        curr ← next
    pi[leader] ← prev
```

Figure 3.2: Reverse the cycle containing *leader*, mutating the permutation.

3.1 Bit Vector

A permutation can easily be inverted in linear time using a linear number of extra bits of space. An n-bit vector can be used to mark corresponding positions in π as they are moved. If processing left to right, the cycle containing position 0 is reversed first marking all positions in the cycle. Then, the lowest position in another cycle is discovered by selecting the next unset bit in the bit vector, i.e., the left most 0 bit. The permutation will be successfully inverted when there are no more unset bits. This is equivalent to using the min leader method described in Chapter 2, but uses $n + \mathcal{O}(\log n)$ extra bits for the bit vector and the constant number of pointers. This has a worst case time of $\mathcal{O}(n)$ because it traverses each cycle exactly one, and traverses the bit vector exactly once. Although this algorithm is not in place, it leads to another algorithm with a sublinear number of extra bits of space.

Following an idea presented in [5], the bit vector can be shrunk by conceptually dividing the permutation into evenly spaced sections. The bit vector can then be applied to one section at a time. This will create a trade-off between the extra space required and the worst case time complexity. If a bit vector v of size $b \leq n$ is used, the permutation can be divided into $\lceil n/b \rceil$ sections of size b (except possibly the last section will be smaller). Conceptually, an outer pointer will iterate through the permutation left to right, testing each position for the min leader. As a cycle is tested, any position found in the current section will be marked in v as known not to be the min leader. When a marked position is found, it can be skipped immediately before testing for leadership. When the outer pointer enters a new section, v will be reset. Figure 3.3 shows the min leader method augmented to utilize the b-bit vector in this way.

```
method isLeader(leader)
    if leader mod b = 0 then
        clear(v)
    if v(leader mod b) = 1 then
        return false
    i ← pi[leader]
    while i > leader do
        if ⌊i/b⌋ = ⌊leader/b⌋ then
            v(i mod b) ← 1
        i ← pi[i]
    return i ≟ leader
```

Figure 3.3: Determine if *leader* is the minimum position in the cycle with the aid of a b-bit vector.

This takes $b + \mathcal{O}(\log n)$ bits for the b-bit vector and the constant number of pointers, and runs in $\mathcal{O}(n^2/b)$ time in the worst case, because it traverses each cycle no more than $\lceil n/b \rceil$ times. The individual cycles' lengths all add up to n.

Theorem 3.1. *In the worst case, the standard representation of a permutation of length n can be replaced with its own inverse in $\mathcal{O}(n^2/b)$ time using $b + \mathcal{O}(\log n)$ extra bits of space.*

A very natural compromise between the space and time trade-off in Theorem 3.1 is given below.

Theorem 3.2. *In the worst case, the standard representation of a permutation of length n can be replaced with its own inverse in $\mathcal{O}(n\sqrt{n})$ time using $\mathcal{O}(\sqrt{n})$ extra bits of space.*

Proof of Theorem 3.2. If the b-bit vector is a $\lfloor \sqrt{n} \rfloor$-bit vector, the proof follows directly from the analysis above. □

To achieve better performance, we need to develop a strategy that will allow us to utilize the Fich et al. leader method to invert a permutation in place.

Chapter 4

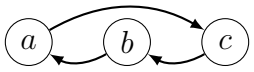
Breaking the Cycle

When a cycle is detected, the naive `invert` algorithm described in the previous chapter will replace the cycle by its reverse. The problem is that this new cycle should not be reversed again, which would occur if the leader of this reversed cycle came later in the scan (i.e. is larger) than that of the original cycle. Figure 4.1 shows a simple example of this: b is the leader of the original cycle, but if reversed, c would be the leader of the new cycle according to the Fich et al. leader. Since $c > b$, the naive algorithm will reverse the cycle once on b and then again on c. Reversing a cycle twice will negate the process of reversing it the first time. We call a cycle with this behaviour a bad cycle.

Definition 4.1. *A* **bad cycle** *is a cycle with the property that if reversed, has a new cycle leader not yet processed, i.e., larger than the original leader.*

A permutation of length n can contain $\Theta(n)$ bad cycles in the worst case. An example of such a permutation is the pattern from our example bad cycle repeated, either concatenated or interwoven, exactly $\lfloor n/3 \rfloor$ times.

Since there is not enough space to use even 1 bit to mark each bad cycle, we need a more clever marking technique. In this chapter, we show how to detect bad cycles and present some techniques to mark such cycles after reversing them.



$a < b < c$

Figure 4.1: An example of a bad cycle.

After reversing a cycle we find the reversed leader and if it is larger than the forward (original) leader, we mark the cycle by breaking it in some way. The idea is to break cycles in such a way that they can be detected and restored easily. To detect whether a cycle has been marked as reversed, we have to detect whether the cycle is broken.

The nature of the Fich et al. leader method naturally reveals the leader of the reverse of the cycle. This allows us to avoid testing each position of the reversed cycle for leadership. If a position i is found to be the leader of the cycle α, then the minimum position in the cycle, regardless of whether it has been reversed or not, is given by $x_d = \alpha_d \circ \alpha_{d-1} \circ \ldots \circ \alpha_0(i)$. The position $j = \alpha_0 \circ \alpha_1 \circ \ldots \circ \alpha_d(x_d)$ is the leader of the reverse of the cycle. Let α^{-1} be the cycle of π^{-1} containing the same positions as α, i.e. the reverse of α. Then j is the leader of α^{-1} because $\alpha_d^{-1} \circ \alpha_{d-1}^{-1} \circ \ldots \circ \alpha_0^{-1}(j) = x_d$. The `isLeader` method in Figure 2.5 will compute $\alpha_d(x_d)$, which will recurse and compute $\alpha_{d-1} \circ \alpha_d(x_d)$, and eventually compute j. The method will store j in $elbow[0]$.

A natural way to break a reversed cycle is to set a position in the cycle to the sentinel value $\emptyset$. The cycle can easily be detected as reversed when $\emptyset$ is encountered by the algorithm. To be capable of storing $\emptyset$, the alphabet needs to be increased by 1. If the permutation length n is a power of 2, this will increase the required word size. If we increase the word size, we might as well use the naive n-bit vector solution. This can be avoided by using 0 to simulate the sentinel value and storing the true position of the 0 pointer in $\mathcal{O}(\log n)$ bits. When traversing a cycle and 0 is encountered, its position can be checked against the true position in constant time to determine if it is meant as a sentinel value or the real 0 pointer. If a sentinel value happens to land on the 0 pointer, the true position can be deleted lazily with 1 bit. Another way to simulate a sentinel value in a non-trivial cycle is to point the element at a position to itself (trivial cycles will never need to be broken). Then the sentinel value can be detected while traversing a cycle by finding a position i such that $i \neq \texttt{pi}[i]$ but $\texttt{pi}[i] = \texttt{pi}[\texttt{pi}[i]]$.

Definition 4.2. *The* **tail** *of a cycle is the predecessor of the leader, i.e., the position i such that $\pi(i)$ is determined to be the leader of the cycle.*

The choice of which position in the reversed cycle to place the (simulated) sentinel value is important. The only position in a cycle stored during the entire traversal is the potential leader, i.e., the current position being tested for leadership. So the sentinel value must be placed at the position that points to the leader, i.e., the tail of the reversed cycle. The tail position can be found by evaluating $\pi(j)$ before reversing the cycle, where j is the leader of the reversed cycle. Mutating the tail turns the cycle into a chain much like a linked list, where the leader is the head and the tail simulates pointing to $\emptyset$.

(a) Cycle restored correctly. (b) Cycle restored incorrectly.

Figure 4.2: Examples of a broken cycle being restored.

When testing a position for leadership and the simulated sentinel value is encountered, the cycle is known to be reversed. The leader method can simulate that the tail points to the potential leader and continue traversing the cycle. If the potential leader is determined to be a leader, the sentinel value is replaced by a pointer to the leader, restoring the cycle.

Figure 4.2 shows two examples of the broken cycle $(x\ b\ y\ a)$ being restored. The solid lines represent what is stored and the dotted lines represent what it will be restored to. In both cases $a < b < \{x, y\}$, so x is the leader of the cycle. In (a), $x < y$, so the cycle is restored correctly because the algorithm finds x before y. But in (b), $y < x$, so the cycle is mutated to a smaller subcycle because the algorithm finds y first. The position y is a valid leader of the subcycle $(y\ a)$. This will happen in any cycle that contains a subcycle with a leader in a smaller position than the leader of the full cycle. So the use of a simple sentinel value fails to restore the cycle in this case. This can be solved by breaking the cycle in different ways that contain more information about the cycle.

4.1 Unique Cycle Lengths

A permutation of length n can contain up to n cycles of length $\ell \geq 1$, but only $c \leq \lfloor\sqrt{2n}\rfloor$ unique cycle lengths. The limited number of unique cycle lengths can be exploited to break and restore bad cycles.

Theorem 4.3. *A permutation of length n has no more than $\lfloor\sqrt{2n}\rfloor$ unique cycle lengths.*

Proof of Theorem 4.3. Assume a permutation of length n has $c \geq \lceil\sqrt{2n}\rceil$ unique cycle lengths. Let ℓ_i be the i-th ranked unique cycle length, where $i \in [c]$. Without loss of generality, assume $\ell_i \geq i + 1$. Then the total length of the permutation must be at least

$$\sum_{i=0}^{c-1} \ell_i \geq \sum_{i=1}^{\lceil\sqrt{2n}\rceil} i \geq \frac{2n + \sqrt{2n}}{2} > n \ ,$$

a contradiction. □

$\pi(i)$	0	1	2
i	4	7	5

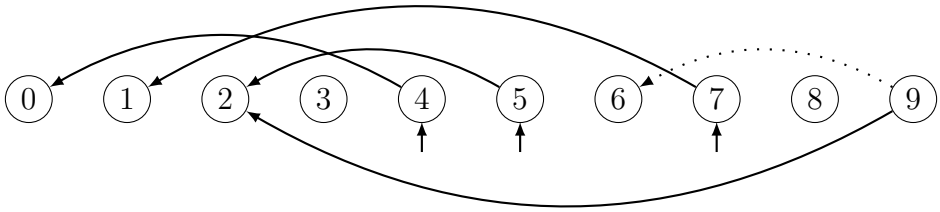


Figure 4.3: An example of a true position table.

The idea is to break bad cycles after reversing them by mutating the tail to point to the rank of the cycles' length, instead of back to the leader. A pointer to the rank of a cycle length can be differentiated from a pointer to another position in the cycle the same way that a 0 pointer can be differentiated from a sentinel value in the previous section, except now there are c pointers to keep track of. The unique cycle lengths are ranked in increasing order of length.

The rank of a cycle length can be obtained by searching a balanced search tree of all the unique cycle lengths. The tree can be built in $\mathcal{O}(n \log n)$ time by iterating over the permutation, inserting the cycle length of every cycle as it is detected using the Fich et al. leader. The tree occupies $\mathcal{O}(\sqrt{n} \log n)$ bits of space and can be searched in $\mathcal{O}(\log n)$ time.

The problem now is that the permutation does not distinguish between pointing to a rank, or pointing to another position in the cycle. Similar to the true position of the 0 pointer in the previous section, a position in the permutation can be distinguished as a mark instead of a pointer to a real position by using a table of true positions. Since there are only c unique cycle lengths, there are only c true positions that must be stored. Those positions are the true positions of the pointers that point to the first c positions.

The table of true positions can be initially set the first time the permutation is traversed, and positions updated as cycles are reversed. The table occupies $\mathcal{O}(\sqrt{n} \log n)$ bits to store exactly c positions. Figure 4.3 shows an example where position 9 is the tail of a marked cycle whose length is ranked 2, i.e., length ℓ_2. Notice the true pointer to position 2 is located at position 5.

When a position i is found such that $\texttt{pi}[i] < c$, then i can be checked against the table in constant time to determine whether it is a true pointer or a mark in constant time. If it is found to be a mark, abort the leader method and do not reverse the cycle. In the case

that the rank of the cycle length and the leader happen to be the same, the corresponding entry in the table can be deleted lazily. Supporting lazy deletion requires only c extra bits.

The cycle can be restored when a mark is found that points to the correct rank of the length of the cycle. This can be checked by searching the tree in $\mathcal{O}(\log n)$ time for the rank of the length traversed so far (including the marked position). The cycle can than be restored by assigning $\texttt{pi}[\mathit{tail}] \leftarrow \mathit{leader}$ where *tail* is the position of the mark, and *leader* is the position currently being tested for leadership. Every marked cycle only needs to be restored once, and will always be restored to the correct value because only the leader can be ℓ steps away from the tail. Every marked cycle will be found because only bad cycles, i.e., cycles that will be found, were marked.

To keep the table of true positions updated only requires $\mathcal{O}(n)$ time in total. It takes $\mathcal{O}(\log n)$ time to search for the rank of a cycle length in the tree. There will be at most 1 search per iteration of the outer loop. The overhead caused by restoring the broken cycles is $\mathcal{O}(n \log n)$ time, the same as running the Fich et al. leader method on every position in the permutation.

Theorem 4.4. *In the worst case, the standard representation of a permutation of length n can be replaced with its own inverse in $\mathcal{O}(n \log n)$ time using $\mathcal{O}(\sqrt{n} \log n)$ extra bits of space.*

This marking technique does not utilize any information from the leader method when restoring cycles. Once the tail of a cycle is encountered the leader method is aborted. The decision to restore the cycle or not is based purely on the length of the cycle traversed.

4.2 The "ɔ" Structure

It is easy to determine if a cycle contains a cycle loop. A *cycle loop* is a subcycle of the original cycle, excluding the original cycle. Following the algorithm described in the previous section, when a cycle is detected it is replaced by its reverse. If the cycle is detected to be a bad cycle, it is broken by having the tail of the reversed cycle point to itself. This creates a loop at the end of the original cycle, transforming it into a structure consisting of a spine with a loop at the end.

Updates to this structure will subsequently be performed. We will call the general structure a "ɔ," (i.e. a σ written backwards). This ɔ starts at the leader and continues through positions in the cycle to the tail. The tail always points to some position within the structure. A graph of this structure is shown in Figure 4.4.

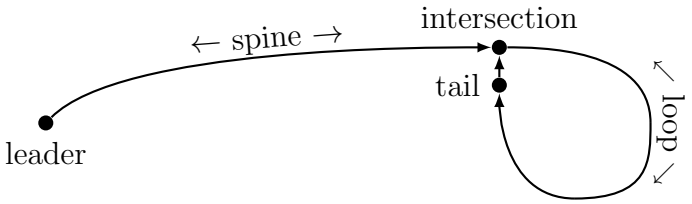


Figure 4.4: The "ɔ" structure consisting of a spine and loop.

Initially $\texttt{pi}[tail] \leftarrow tail$, Figure 4.5 shows an example of this. Position a is the leader, g is both the tail and the intersection, $(a \ \ldots \ f)$ is the spine and (g) is the loop. The solid arrows represent what is stored, and dotted arrow represents the real cycle.

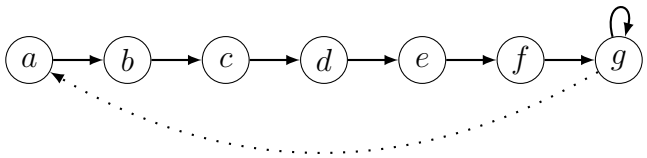


Figure 4.5: A reversed cycle marked with a trivial loop.

When everything has been completed, $\texttt{pi}[tail]$ will point to the leader. In between, $\texttt{pi}[tail]$ is the erroneous leader of the ɔ, i.e. would be the leader of a cycle if $\pi(tail)$ pointed to that position. For example, in Figure 4.6 the position d is a valid leader of what appears to be a complete cycle, and g remains the tail to both the real leader a and the intersection d. Now the spine has shrunk to $(a\ b\ c)$ and the loop expanded to $(d \ \ldots \ g)$.

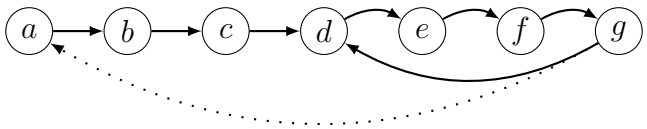


Figure 4.6: An intermediate step in the restoration of the cycle.

The process of restoring such broken cycles is performed by interleaving the scan to determine whether a position is a cycle leader with two other scans which determine the tail. Since we don't know whether a cycle is in proper or broken form we must perform this interleaved process. We call the three scans $\mathcal{F}$, $\mathcal{T}$ and $\mathcal{H}$. We use $\mathcal{F}$ to determine whether a position is a cycle leader, using the Fich et al. leader method. As the cycle could indeed

be broken, this process proceeds the slowest, at 1 step per call. We use $\mathcal{T}$ and $\mathcal{H}$ to detect and find the tail of a broken cycle. This process is done by coordinating the two scans: $\mathcal{T}$ (for tortoise), which proceeds at 2 steps per call; and $\mathcal{H}$ (for hare), which proceeds at 4 steps per call.

The $\mathcal{F}$ scan will terminate on one of the following three cases:

- The first case is $\mathcal{F}$ determines the position is not a cycle leader. If so, the entire process of all three scans is aborted.
- The second case is $\mathcal{F}$ determines the position is a cycle leader. If so, the reversed (and perhaps broken) cycle replaces the current cycle.
- The third case is $\mathcal{F}$ determines that its evidence is consistent with the hypothesis that the position is the leader of a cycle whose tail is a position already determined by $\mathcal{T}$ and $\mathcal{H}$. If so, mutate the tail to point to the current position, enlarging the loop of the $\mathfrak{o}$ and potentially eliminating the spine.

The $\mathcal{T}$ and $\mathcal{H}$ scans have two phases:

- The first phase is to detect if the cycle is broken. There are two cases.
  - If $\mathcal{H}$ returns to the starting position, the cycle is not broken so $\mathcal{T}$ and $\mathcal{H}$ are aborted and $\mathcal{F}$ continues until one of its first two cases happen.
  - Otherwise, $\mathcal{T}$ and $\mathcal{H}$ meet at some position, i.e. there is a position in common in the 4:2 (or possibly 2:1) steps taken in the current call. If this happens, we know we have a broken cycle and advance to the second phase.
- The second phase is to find the tail of the cycle. Reset $\mathcal{T}$ to the starting position and decrease the speed of $\mathcal{H}$ to 2 steps per call. When $\mathcal{T}$ and $\mathcal{H}$ meet for a second time, we know the length of the spine of the $\mathfrak{o}$ and can identified the tail. $\mathcal{F}$ is left to continue, either to abort from a position that is determined not to be a cycle leader or to increase the size of the loop of the $\mathfrak{o}$.

The $\mathcal{F}$ scan proceeds similar to the previous section. Bad cycles are detected exactly the same way. The major difference is that bad cycles are broken by assigning $\texttt{pi}[\mathit{tail}] \leftarrow \mathit{tail}$.

Figure 4.7 shows a method to do one iteration of this cycle loop detection process. This method is intended to be called twice to represent one call of the $\mathcal{T}$ and $\mathcal{H}$ scans.

The cycle loops are detected using the classic "the tortoise and the hare" algorithm [8]. That is two pointers, t and h, initialized to the potential leader. For every iteration along the cycle, t advances 1 step and h advances 2 steps. If the cycle contains a loop, the pointers t and h will rendezvous inside the loop in less than ℓ iterations. This is because at each iteration, the forward distance between the two pointers is decreased by 1. If the cycle is not marked, h will wrap around and reach the potential leader again in ℓ (or $\ell/2$ if ℓ is even) iterations. Notice it is possible for h to pass t without meeting. This represents the first phase of the $\mathcal{T}$ and $\mathcal{H}$ scans.

If the cycle contains a cycle loop, the t pointer is moved back to the potential leader. The h pointer remains at the rendezvous. Both pointers traverse the cycle, one step per iteration until they meet for a second time. This second meeting position is the *intersection* of the cycle, the position with two in pointers. This represents the second phase of the $\mathcal{T}$ and $\mathcal{H}$ scans.

```
method detectCycleLoop()
    switch state do
        case SEARCHING_FOR_LOOP
            t ← pi[t]
            h ← pi[pi[h]]
            if t = h then
                state ← LOOP_DETECTED
                t ← pi[t]
            if h = leader then
                state ← CYCLE_NOT_MARKED
        case LOOP_DETECTED
            if t = pi[h] then
                tail ← h
                state ← IDENTIFIED_TAIL
            t ← pi[t]
            h ← pi[h]
```

Figure 4.7: One iteration of the cycle loop detection and tail identification.

When a cycle loop is detected, we are interested in finding the tail of the cycle, not the intersection. So in the second phase, the method replaces h with $\pi(h)$ and treats that like the pointer. The t pointer is advanced by one to compensate. When the second phase is complete, $\pi(h)$ will point to the intersection and h will point to the tail.

Theorem 4.5. *The tail of a broken cycle can be identified in less than 2ℓ iterations.*

Let ℓ be the length of the cycle and λ be the length of the cycle loop. Let μ be the distance from the potential leader to the intersection and δ be the distance from the intersection to the first rendezvous. In the first phase of the cycle loop detection, the t pointer travelled distance $d_t = \mu + \delta$ and the h pointer travelled distance $d_h = \mu + k\lambda + \delta$ where $k \in \mathbb{Z}^+$. This is because t did not wrap around the loop and h had to wrap around at least once to catch up to t from behind. Since t travelled twice as fast as h, we know

$$\begin{aligned} 2d_t &= d_h \\ 2(\mu + \delta) &= \mu + k\lambda + \delta \\ \mu &= k\lambda - \delta \ . \end{aligned}$$

The t pointer will travel μ steps and intersect the h pointer. The h pointer wrapped around the loop some number of times, but stopped δ steps short of the rendezvous, i.e., at the intersection.

Proof of Theorem 4.5. Since $d_t < \ell$ is the number of iterations of the first phase and $\mu < \ell$ is the number of iterations of the second phase, the total number of iterations taken by the loop detection process is $d_t + \mu < 2\ell$. □

Figure 4.8 shows how to run the cycle loop detection and the Fich et al. leader methods in parallel, such that the loop detection will find the tail before the leader method reads it. This method is intended to replace the call to `next()` from Figure 2.8. The Fich et al. leader method advances along the cycle only by calling `next()`. This represents interleaving the $\mathcal{F}$, $\mathcal{T}$ and $\mathcal{H}$ scans exactly.

```
method next()
    detectCycleLoop()
    detectCycleLoop()
    elbow[0] ← pi[elbow[1]]
    if state = IDENTIFIED_TAIL and elbow[1] = tail then
        elbow[0] ← leader
```

Figure 4.8: Determine the next position in the cycle, compensating for marked cycles.

The final algorithm to invert a permutation in place is shown in Figure 4.9. If the Fich et al. leader method, `isLeader` as shown in Figure 2.5, determines a potential

leader not to be a leader, it will stop calling `next()` thus terminating all scans. That the entire process takes time $\Theta(\mathcal{F})$ follows from the interleaving. Running the loop detection and leader methods in parallel increases the run time by only a constant factor. The loop detection method requires only a constant number of pointers. Since the Fich et al. leader method requires $\mathcal{O}\left(\log^2 n\right)$ bits, this does not increase the space complexity.

```
method invert()
    for leader ← 0 to n − 1 do
        state ← SEARCHING_FOR_LOOP
        t ← leader
        h ← leader
        if isLeader(leader) then
            newLeader = elbow[0]
            switch state do
                case IDENTIFIED_TAIL
                    pi[tail] ← leader
                case CYCLE_NOT_MARKED
                    if newLeader > leader then
                        newTail ← pi[newLeader]
                        reverse(leader)
                        pi[newTail] ← newTail
                    else
                        reverse(leader)
```

Figure 4.9: Algorithm to invert a permutation in place.

There is an invariance that at every step, the tail of the $\mathfrak{o}$ structure is the tail of the cycle loop. The new leaders will be found in monotonically increasing order of position. As new leaders are found, the length of the loop is increased and the length of the spine is decreased. Each new leader is found outside of the loop because the leader of the loop has already been processed. When the entire permutation has been iterated over, every leader has been found and every cycle has been fully restored. Therefore the permutation will be correctly inverted. Theorem 4.4 can be strengthened to our main result.

Theorem 4.6. *In the worst case, the standard representation of a permutation of length n can be replaced with its own inverse in $\mathcal{O}(n \log n)$ time using $\mathcal{O}\left(\log^2 n\right)$ extra bits of space.*

Chapter 5

Conclusion and Future Work

The algorithm presented in Section 4.2 to invert a permutation in place can easily be adapted to utilize any leader method. A better leader method will yield a better invert algorithm without adding to the worst case time or space complexity. The overhead added by using the cycle loop detection technique is only a constant number of pointers and linear time. Since any leader method must look at every position and use at least a constant number of pointers, there is no leader method that will use less resources than the cycle loop detection technique.

A specific property of the Fich et al. leader method was exploited to detect bad cycles easily. Bad cycles can detected by any leader method by reversing the cycle then traversing it, testing each position for leadership until the leader of the reversed cycle is found. In the worst case, this will double the time spent on the leader method because every position will be tested for leadership no more than twice. The tail of the reversed cycle can be found in a similar way.

The bit vector algorithms presented in Chapter 3 are equivalent to using the min leader. The cycle leader algorithms presented in this thesis always process the permutation left to right, but that is not a requirement. It is not known if every algorithm to invert (or apply) a permutation can be expressed in terms of a leader method.

5.1 Future Work

Future work could be done on applying other transformations to the array representation of a permutation in place. For example, apply an arbitrary power to a permutation.

That is, replace the standard representation of π in `pi` by the representation of π^q, where

$$\pi^q(i) = \begin{cases} \pi \circ \pi^{q-1}(i) & \text{if } q > 0 \\ i & \text{if } q = 0 \\ \pi^{-1} \circ \pi^{q+1}(i) & \text{if } q < 0 \end{cases} \tag{5.1}$$

for any arbitrary power $q \in \mathbb{Z}$. Inverting a permutation is the special case when $q = -1$.

Fich et al. [5] presented an algorithm to apply an arbitrary power of a permutation to an array in place. The algorithm is an extension of the permute algorithm presented in Chapter 2, but does not increase the time or space complexity. This is interesting because the time and space requirements do not depend on q. The only difference between applying an arbitrary power q and applying the original permutation is the `rotate` method. Instead of rotating the cycle one step, rotate it (q mod ℓ) steps, where ℓ is the length of the cycle. Rotating a cycle several steps forward is essentially the same as transposing an array using the method discussed in Section 4.10 of [2].

In terms of applying the arbitrary power q to the permutation π, the cycles need to be rotated in place, mutating `pi`. Equation (5.1) shows the definition of the power of a permutation. The problem could be done for positive q by rotating each cycle in π, $q - 1$ steps forward. For negative q, the permutation can be inverted first. An issue to deal with is the fact that the leader can change, just like in bad cycles in the previous chapter. Additionally, a cycle may be split into several cycles by rotating it several steps. Figure 5.1 shows an example of an even length cycle being squared and split into two new cycles.

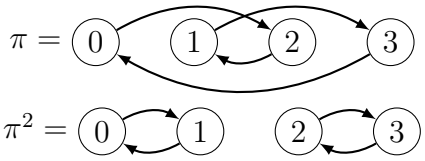


Figure 5.1: An example of a split cycle.

The split cycles issue can be avoided by using max leader, i.e., the maximum position in the cycle as leader. This way the smaller cycles will never be encountered when iterating over the rest of the permutation. But as discussed in terms of min leader in Chapter 2, this will be too slow. Future work can be done to develop a technique similar to that in Section 4.2 that will allow the use of the Fich et al. leader for applying arbitrary powers to permutations.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co., 1974.

[2] Gilles Brassard and Paul Bratley. *Algorithmics: Theory & Practice*. Prentice-Hall, Inc., 1988.

[3] Mariano Paulo Consens and Timothy Snider. Maintaining very large indexes supporting efficient relational querying, August 14 2001. US Patent 6,275,822.

[4] Danny Dolev, Maria Klawe, and Michael Rodeh. An $\mathcal{O}(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, 1982.

[5] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[6] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[7] Daniel S. Hirschberg and James Bartlett Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969.

[9] J. Ian Munro and Venkatesh Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[10] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Satti Srinivasa Rao. Succinct representations of permutations. In *Automata, Languages and Programming*, pages 345–356. Springer, 2003.

[11] Gary L. Peterson. An $\mathcal{O}(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4(4):758–762, 1982.